CS3.301 Operating Systems and Networks

Memory Virtualization - Paging: Mechanisms and Policies

Karthik Vaidhyanathan

https://karthikvaidhyanathan.com

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
HYDERABAD

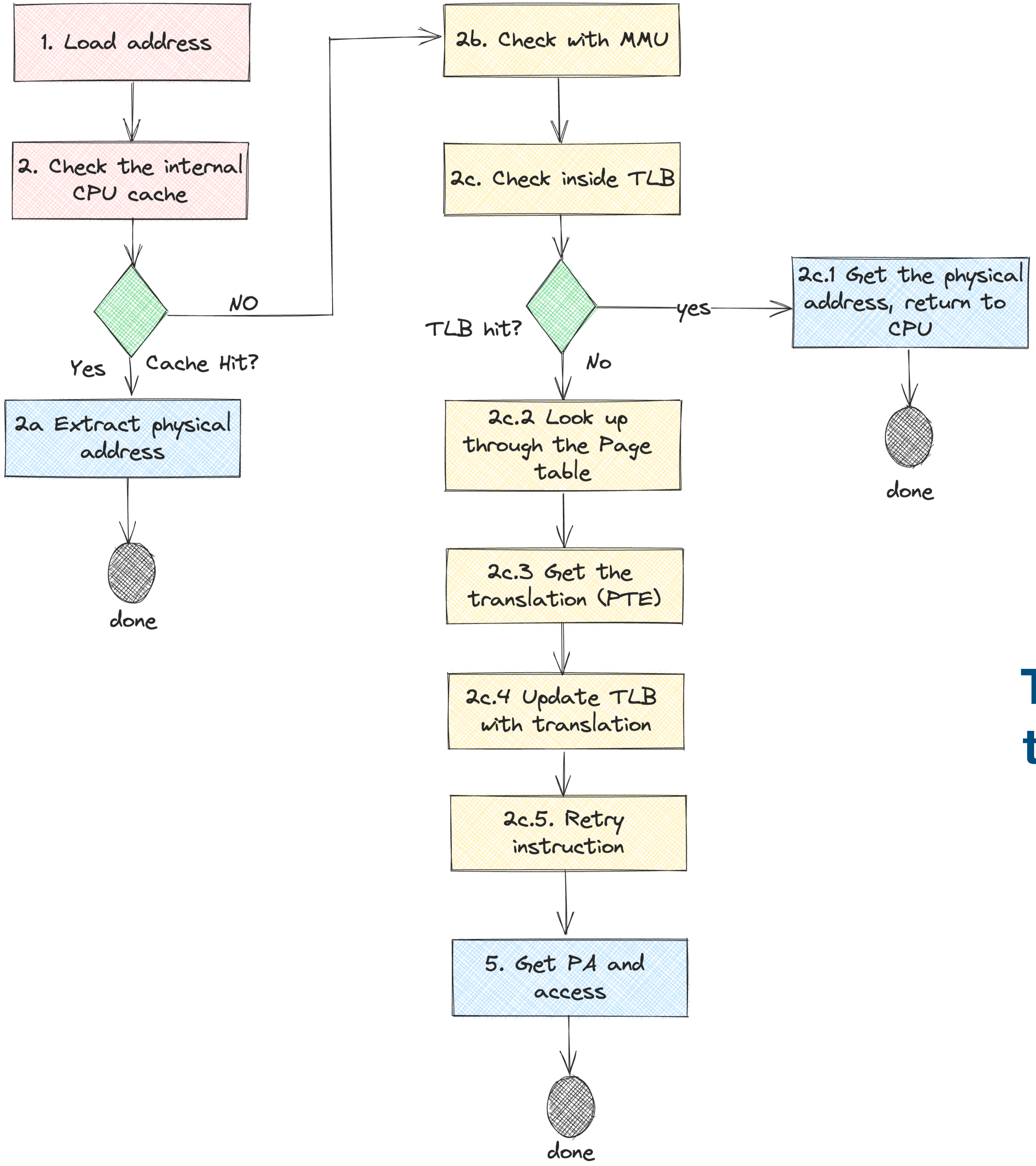
Software Engineering Research Centre

Acknowledgement

The materials used in this presentation have been gathered/adapted/generate from various sources as well as based on my own experiences and knowledge -- Karthik Vaidhyanathan

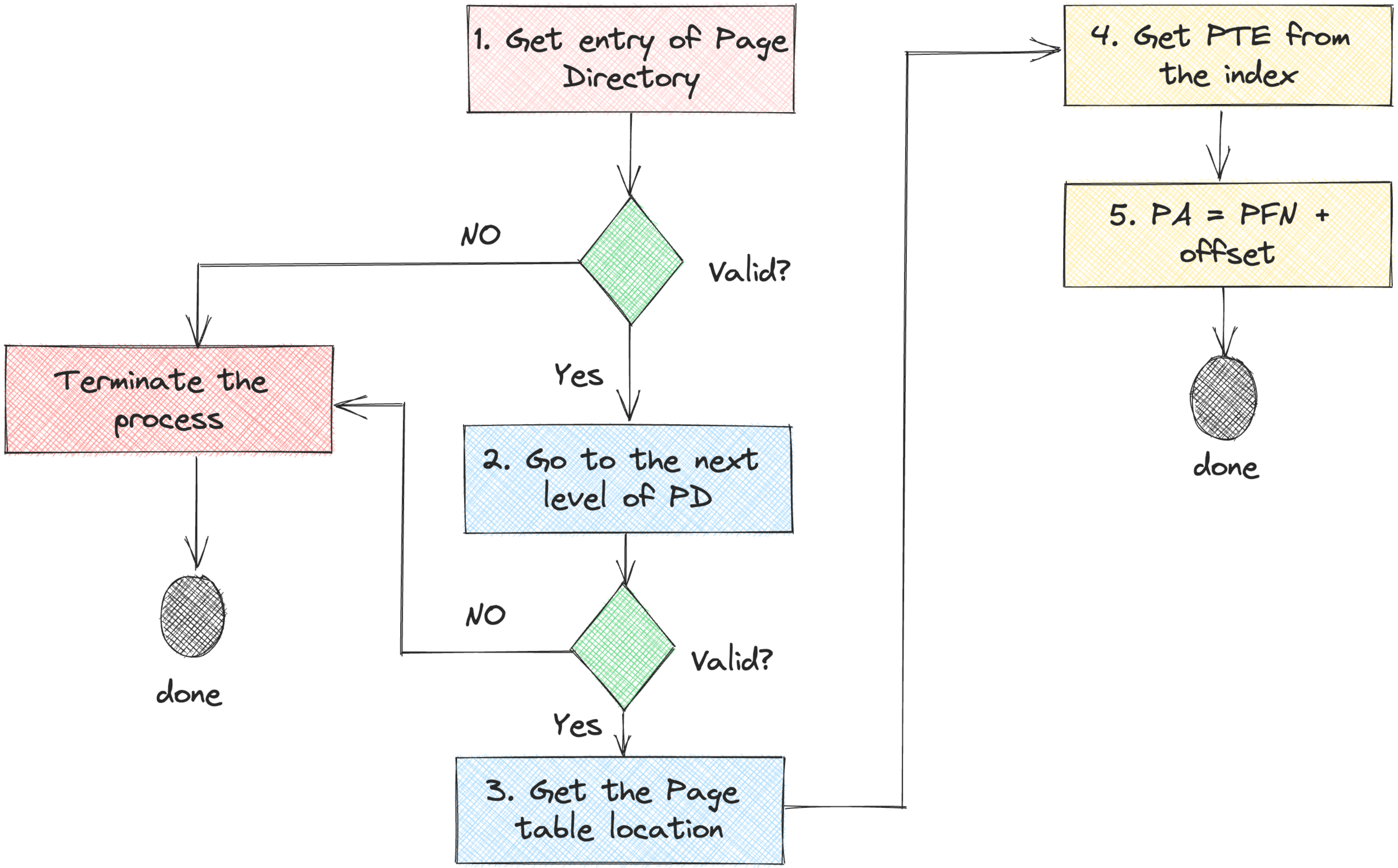
Sources:

- Operating Systems: In three easy pieces, by Remzi et al.
- Lectures on Operating Systems by Youjip Won, Hanyang University



The overall address translation Process

The overall Flow of Multi-level page tables



Some more questions needs answers!

- What if the size of the address space is larger than the physical space?

- Are all pages of all active processes always in main memory?
  - All of them may not fit in the main memory - so how does it work?

- OS uses a part of the disk (swap space) to store pages that are not active in use - How does that work?

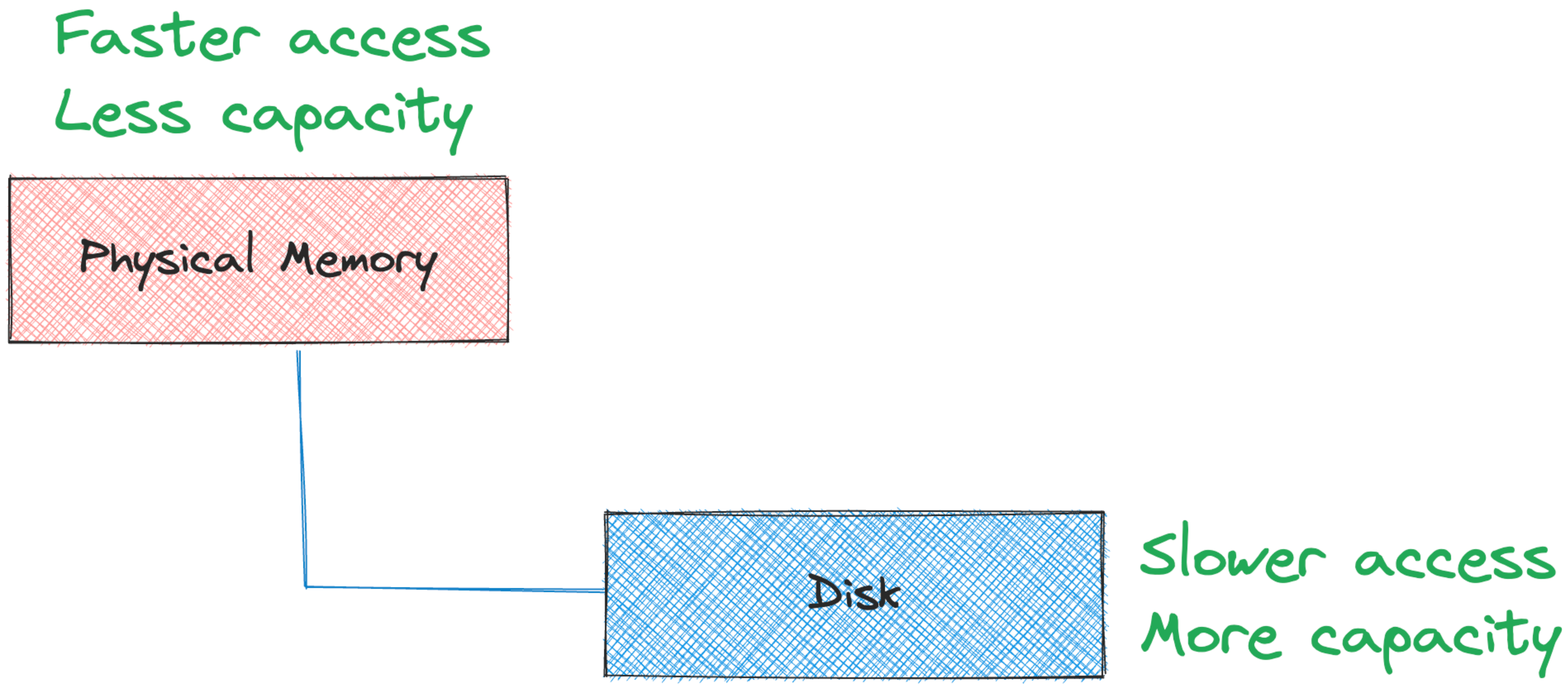
What about Memory Hierarchy?

- Not every page needs to be available in physical memory
  - Use additional level of memory
  - OS can stash away portions of address spaces that are currently not in demand
  - What is something that has more memory than physical memory?
    - What about hard disk? But how?

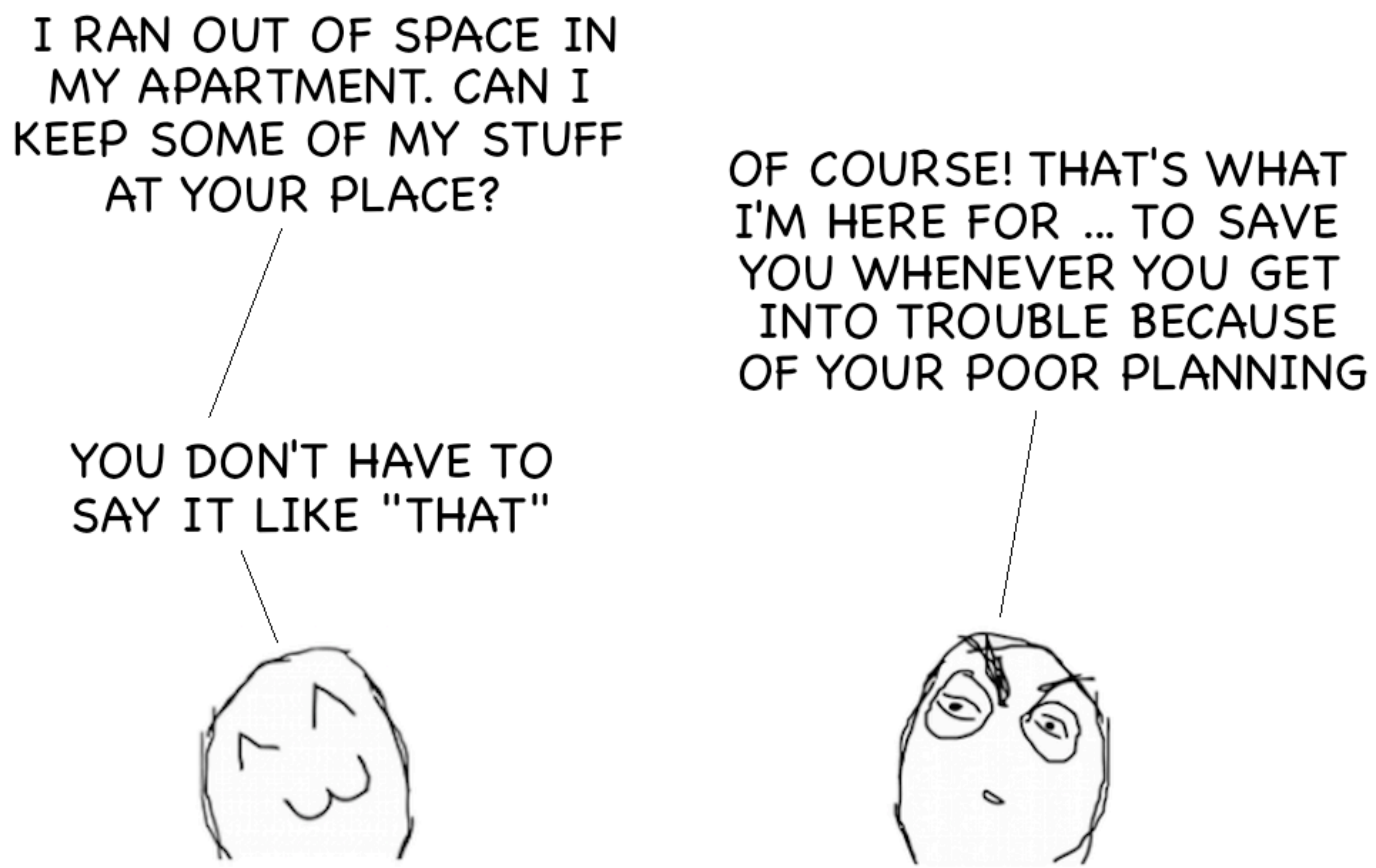
Moving between Physical Memory and Disk

Demand Paging

- How can OS make use of the slower, larger and faster, smaller device more effectively?
- This allows to provide the illusion of large virtual address space
- Think about manually moving pieces of code and data in and out of memory as needed



Leverage Swap Space!!

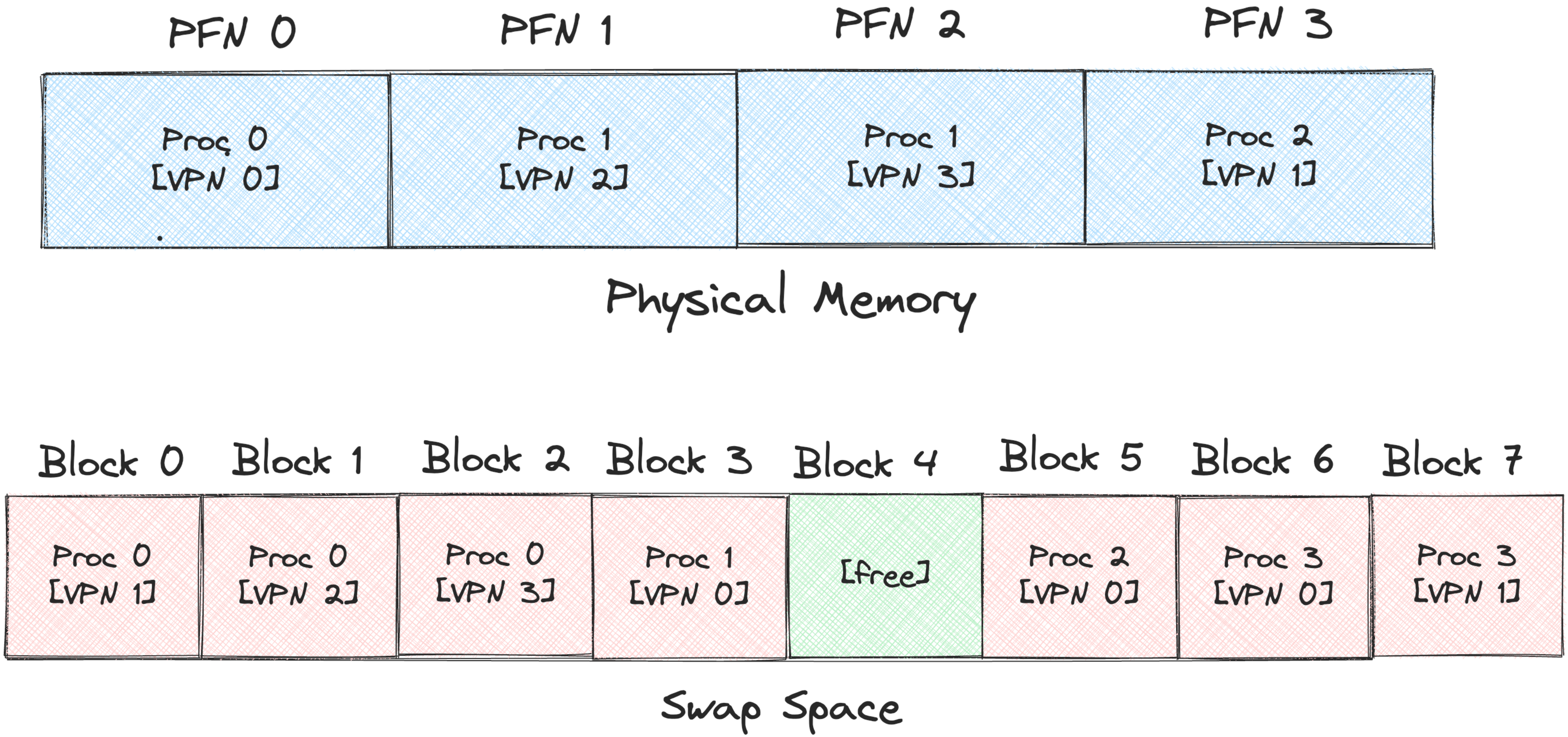


https://prateekvjoshi.com/2015/03/21/how-to-add-swap-space-on-ubuntu/

Leverage Swap Space

- Enables OS to swap out some pages to **swap space (in disk)**
- This is much needed in multi-programming
- Physical memory cannot hold all the pages - Always limited
- **Swap Space:** Dedicated space in the memory which can be used by OS to swap in and out pages
- OS should be able to read and write page sized units from/to swap space
- How does OS remember the location of page in swap space?

Swap Spaces



PFN 0
PFN 1
PFN 2
PFN 3
Proc 0 [VPN 0]
Proc 1 [VPN 2]
Proc 1 [VPN 3]
Proc 2 [VPN 1]
Physical Memory
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
Proc 0 [VPN 1]
Proc 0 [VPN 2]
Proc 0 [VPN 3]
Proc 1 [VPN 0]
[free]
Proc 2 [VPN 0]
Proc 3 [VPN 0]
Proc 3 [VPN 1]
Swap Space

Swap Spaces

- The use of swap space allows OS to give perception that process has abundant memory
- OS can take from memory and add it to swap space
  - Swapping provides a big support to OS for memory management
  - How to make use of swap?
    - When to use swap?
    - How to make memory management work?

Can we not leverage present bit?

- In usual scenario, if there is TLB miss
  - OS gets the PTE from the page table
  - But if we need to use swap
    - The OS needs to know if the page is in the physical memory
    - Leverages the use of **present bit**
    - If present bit is 1, page is in the page table
    - If present bit is 0, page is in the swap space

Page Fault

- The act of accessing page that is not there in the physical memory
- When a page is not in the physical memory
  - Hardware does not know how to handle it, raises exception
  - The OS has to service the page fault
  - Piece of code to achieve this - **Page Fault Handler**
  - This needs to be done both in the case of hardware of software-managed TLB

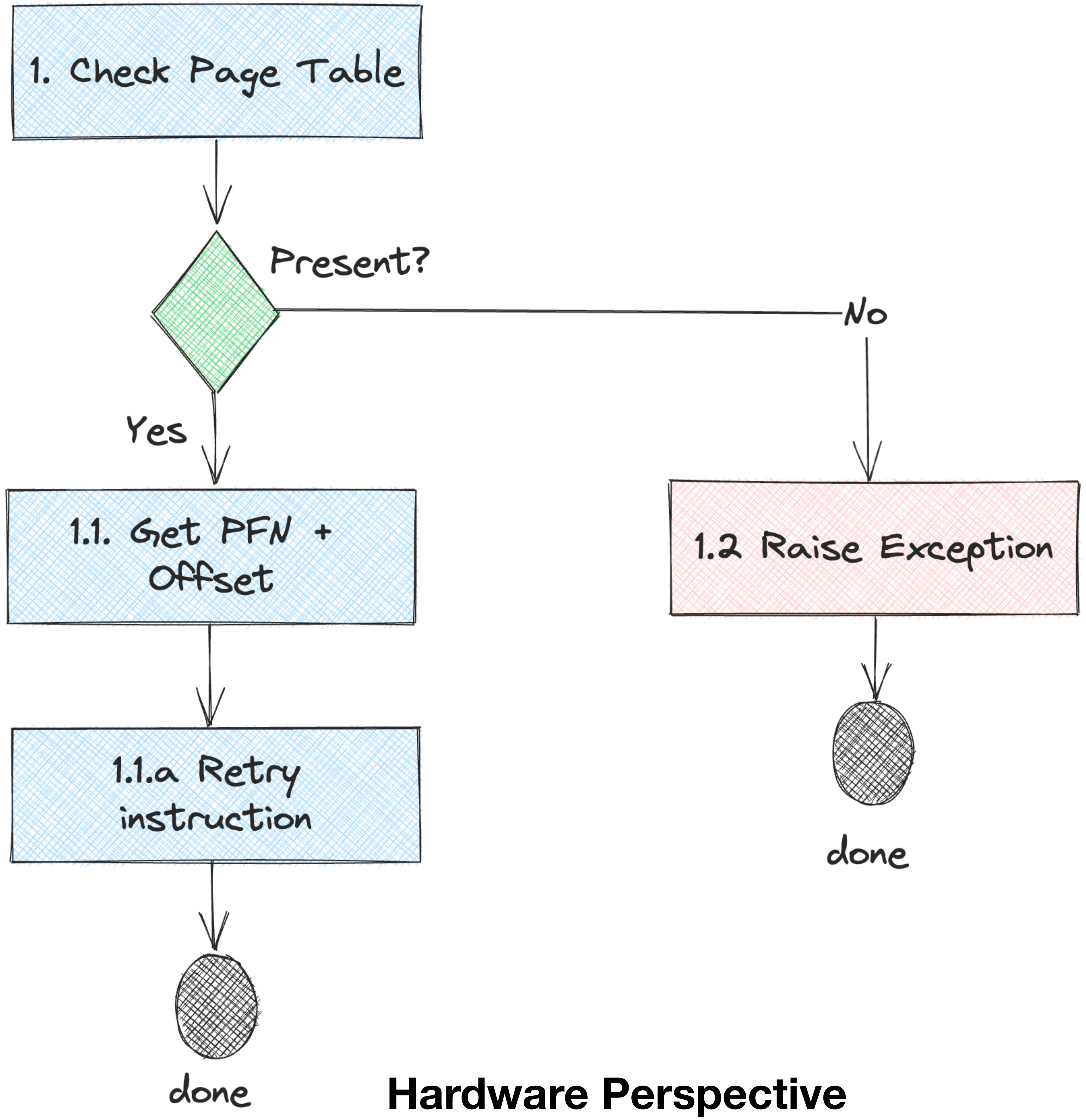
Page Fault

- How does OS know where to find a page during page fault?
  - OS can make use of bits in PTE (PFN) to store such information
  - On page fault
    - OS searches through PTEs
    - Gets the address from the PFN
    - Page fault handling involves Disk I/O
    - There is a possibility for **Context-switch** - How?

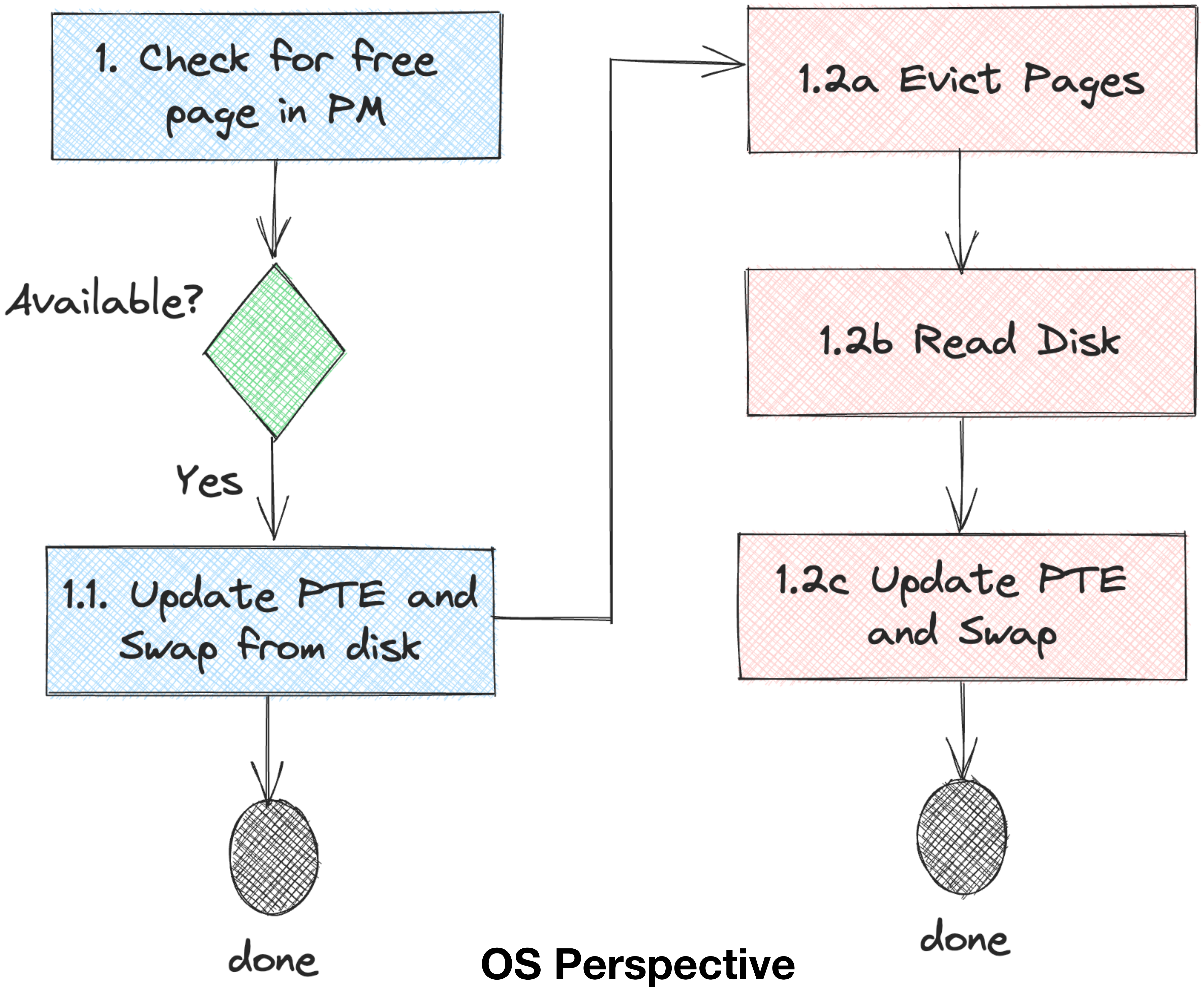
Page Fault and Context Switches

- On page fault
  - OS has to handle the fault
  - The process will be moved to a **blocked** state
  - OS will be free to run other process
  - On the servicing of page fault -> Another process can be executed
- What if the memory is full?
  - Some pages has to be moved out of physical memory
  - The process of swapping pages in/out from/of memory - **Page Replacement!**

Page Fault Control Flow



Hardware Perspective



OS Perspective

Page Replacement Policy

- How can the OS decide which **pages to evict** form the memory?
  - Makes use of a replacement policy
- Goal is to **maximize page access** from memory
  - If we treat physical (main) memory as cache
  - Goal is to **minimize cache misses**
  - In other words, goal is to maximize number of times page is read from/ written to in main memory

Defining a metric

- % of hits to cache = Cache Hit
- % of misses on cache = Cache Miss
- Average Memory Access Time (AMAT)
- $AMAT = (Hit\% * T_M) + (Miss\% * T_D)$
- T_M - Cost of accessing Memory
- T_D - Cost of accessing Disk

A Small Illustration

- Consider a tiny address space of 4 KB with 16 pages
  - Page size: 256 bytes; VPN: 4 bits and offset: 8 bits
  - Assume that every page except Page 3 are in virtual memory
  - Sequence: hit, hit, hit, miss, hit, hit, hit, hit, hit, hit
  - Hit% = 90, 10% mis rate
  - AMAT = 0.9 * 100 (nano seconds) + 0.1*10 (microseconds) ~ **1 ms**
  - **What if hit rate is 99.9%?**
  - **The cost of disk access can be so high - A single miss is very costly!!**

Optimal Replacement Policy

- Developed by Belady, many years ago, also known as MIN
- Simple approach - replace the page that will be accessed farthest in future
- If some page needs to be evicted
  - Evict the page that is needed farthest from now
  - Pages in the cache are more important now than the pages that will be access farthest in future
- Known as the optimal policy - Difficult to implement!

Optimal Replacement Policy

- Consider a stream of Virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1 cache size: 3

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	**2**	0, 1, 3
0	Hit		0, 1, 3
3	Hit		0, 1, 3
1	Hit		0, 1, 3
2	Miss	**3 (0 or 3)**	0, 1, 2
1	Hit		0, 1, 2

Hits: 6/11

Optimal Replacement Policy

- 2 is replaced first in the example as it is required least in future compared to others
- The first three access are misses - Cache is in empty state
  - This is called cold-start miss or compulsory miss
- Optimal is like an ideal policy for any set of access
- Very difficult to implement as in reality future is not known apriori!
- What can be a good alternative?

FIFO Policy
First in First Out

- Earlier systems avoided the complexity to implement the optimal policy
- Simple first in first out approach
- Pages are placed in a queue
    - When need for eviction: Evict the pages on the top
- Advantage: Very easy to implement
- At any point, evict the one that came first!

FIFO Policy

- Consider access to a stream of Virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1, cache size: 3 pages

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0,1
2	Miss		0, 1, 2
0	Hit		-> 0, 1, 2
1	Hit		-> 0, 1, 2
3	Miss	**0**	-> 1, 2, 3
0	Miss	**1**	-> 2, 3, 0
3	Hit		-> 2, 3, 0
1	Miss	**2**	-> 3, 0, 1
2	Miss	**3**	-> 0, 1, 2
1	Hit		-> 0, 1, 2

Hits: 4/11

Needs Improvement !!

Belady’s Anomaly

- Consider a stream 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, cache size: 3

Access	Hit/Miss?	Evict	Cache State
1	Miss		1
2	Miss		1, 2
3	Miss		1, 2, 3
4	Miss	1	2, 3, 4
1	Miss	2	3, 4, 1
2	Miss	**3**	4, 1, 2
5	Miss	4	1, 2, 5
1	Hit		1, 2, 5
2	Hit		1, 2, 5
3	Miss	**1**	2, 5,3
4	Miss	2	5, 3, 4
5	Hit		5, 3, 4

Hits: 3/12

What if Cache size is 4?

Belady’s Anomaly

- Consider a stream 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, cache size: 4

Access	Hit/Miss?	Evict		Cache State
1	Miss			1
2	Miss			1, 2
3	Miss			1, 2, 3
4	Miss			1, 2, 3, 4
1	**Hit**			1, 2, 3, 4
2	**Hit**			1, 2, 3, 4
5	Miss	1		2, 3, 4, 5
1	Miss	2		3, 4, 5, 1
2	Miss	3		4, 5, 1, 2
3	Miss	**4**		5, 1, 2, 3
4	Miss	5		1, 2, 3, 4
5	Miss	1		2, 3, 4, 5

Hits: 2/12

Something strange?

Random Policy

- Picks a random page to replace under memory pressure
- Has properties similar to FIFO - very easy to implement
- Random totally depends on luck to get it right
- Need to run multiple times to get good approximation
- May perform a touch better than FIFO but little less than optimal
- How does that work?

Random Policy

- Consider access to a stream of Virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1, cache size: 3 pages

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		0, 1, 2
1	Hit		0, 1, 2
3	Miss	**1**	0, 2, 3
0	Hit		0, 2, 3
3	Hit		0, 2, 3
1	Miss	**2**	0, 3, 1
2	Miss	**0**	3, 1, 2
1	Hit		3, 1, 2

Hits: 5/11

Least Recently Used (LRU)

- As done in scheduling can we use history as a guide?
- **Idea:** If a page was referenced recently, it may be likely to be referenced again
- The historical information that page replacement can use: **Frequency**
- The more recently a page has been accessed, it should not be replaced soon
- **LRU:** replaces the least recently used page
- Works well due to locality of references (Temporal locality here)

Least Recently Used

- Consider access to a stream of Virtual pages: 0, 1, 2, 0, 1, 3, 0, 3, 1, 2, 1, cache size: 3 pages

Access	Hit/Miss?	Evict	Cache State
0	Miss		0
1	Miss		0, 1
2	Miss		0, 1, 2
0	Hit		1, 2, 0
1	Hit		LRU -> 2, 0, 1
3	Miss	**2**	LRU -> 0, 1, 3
0	Hit		LRU -> 1, 3, 0
3	Hit		LRU -> 1, 0, 3
1	Hit		LRU -> 0, 3, 1
2	Miss	**0**	LRU -> 3, 1, 2
1	Hit		LRU -> 3, 2, 1

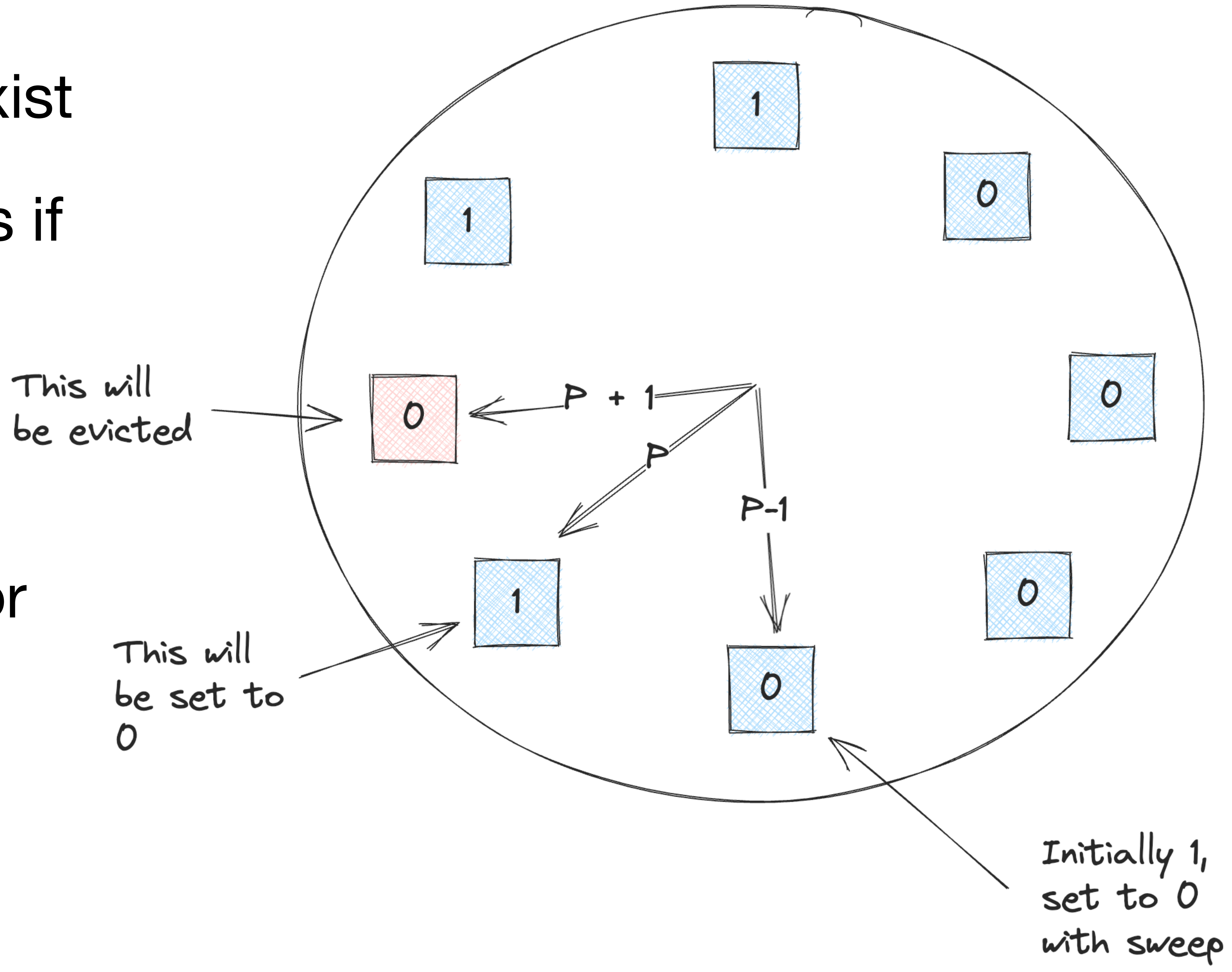
Hits: 6/11

Implementing LRU

- How to implement policies like LRU?
- OS is not involved in every memory access - How does it know which page is the least recently used?
- Hardware can help along with some approximations
- When a page is accessed, MMU can set the “accessed” bit to 1 in PTE
  - It is the responsibility of OS to clear the bit
- How can the “accessed” bit be used by the OS to implement LRU?

Implementing LRU

- Simple and early approach, many approaches exist
- When replacement needs to be done, OS checks if a page, P pointed to has use bit 1 or 0
    - If use bit is 1, set that to 0 and go to the next page
    - If use bit is 0, that page becomes the victim for eviction
    - Avoid repetitive scanning, not the optimal, still works better
- Improved version makes use of **dirty bit**

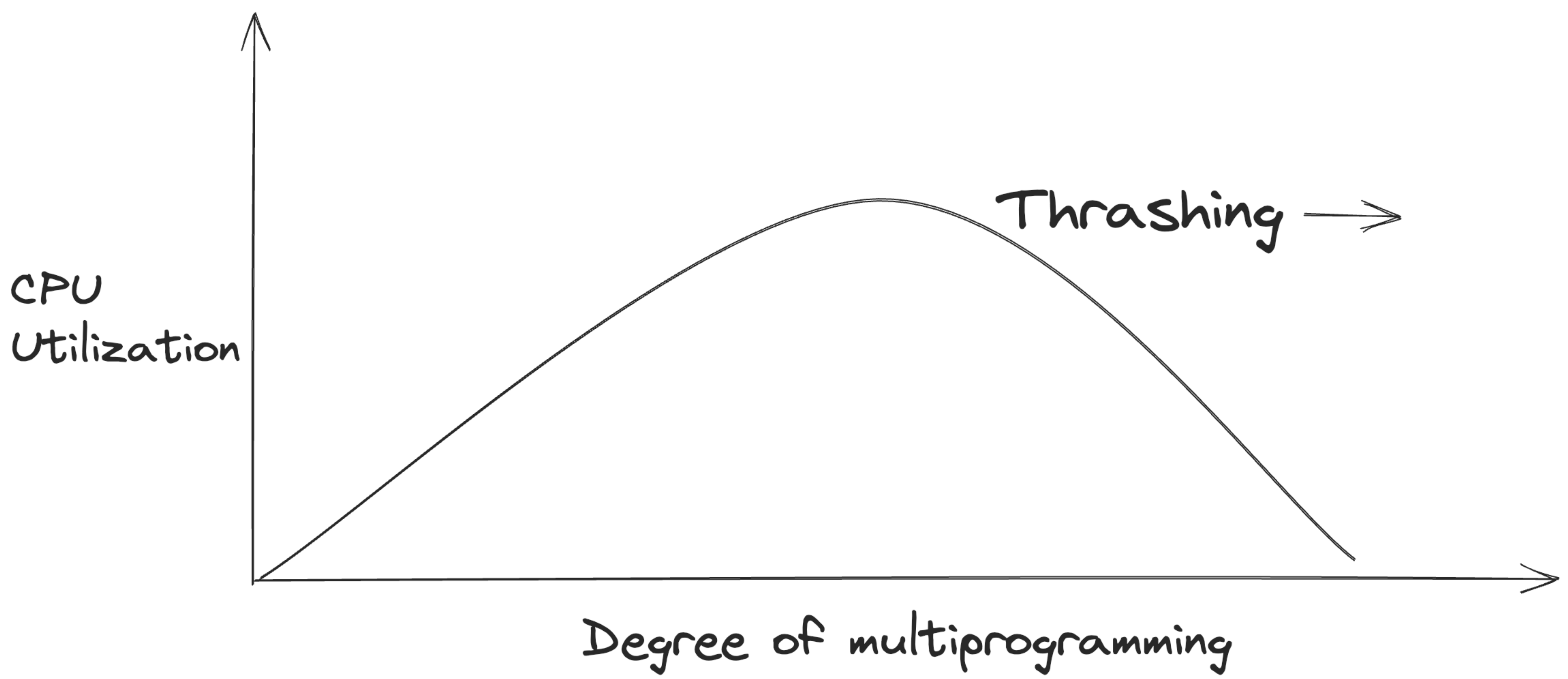


Implementing LRU

- If a page has been modified
  - It has to be pushed to disk and write to disk needs to be done
  - If not, if page is clean, it can simply be replaced - overhead is less!
- Make use of a new bit - modified or dirty bit in the PTE
  - If a page has been recently modified -> dirty bit is set to 1
  - If the page is clean and not modified -> dirty bit is set to 0
- Clock algorithm first scans for pages that are both unused and clean
- If no page with both unused and clean status, evict unused and dirty pages

Thrashing

- What to do if the memory is oversubscribed?
  - Memory demands of running processes exceeds available physical memory
  - System will be constantly paging - Thrashing
  - One approach is admission control
  - Try to run only a subset of processes instead of trying to accommodate everything
  - More aggressive approach: Kill memory intensive processes



SCAN ME

Thank you

Course site: karthikv1392.github.io/cs3301_osn
Email: karthik.vaidhyanathan@iiit.ac.in
Twitter: @karthi_ishere